RICE UNIVERSITY

Type-Based Specialization in a Telescoping Compiler for Matlab

by

Cheryl McCosh

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Ken Kennedy, Professor, Chair
Computer Science

Keith Cooper, Professor
Computer Science

Dan Sorensen, Professor
Computational and Applied Mathematics

Houston, Texas

Nov, 2002

© Copyright

Cheryl McCosh

2003

Type-Based Specialization in a Telescoping Compiler for Matlab

Cheryl McCosh

Abstract

This thesis develops telescoping-language technology for automatically generating high performance libraries from development code written in high-level languages, like Matlab. The generated library subroutines have pre-optimized variants based on possible uses of the library. Specifically, a specialized variant is generated for each possible type configuration on inputs to the library. This thesis develops an efficient graph-theoretical, constraint-based algorithm for inferring types in Matlab needed for translation into lower-level languages, optimization, and determination of which specialized variants to generate. The algorithm computes type jump functions, which allows it to infer types interprocedurally.

To illustrate the power of the technology, this thesis develops ARGen, a system which generates code equivalent to ARPACK, a Fortran linear algebra library, from its Matlab development code.

By accomplishing the type inferencing algorithm and ARGen, which uses it, this thesis provides a basis for a general system for telescoping languages.

Acknowledgments

I would first like to thank my advisor, Prof Ken Kennedy, for his support, input, and for helping me build confidence in this field. Prof Dan Sorensen provided the motivation behind this work and the ARPACK code as well as guidance and help in understanding the problem. Prof Keith Cooper was alway available for advice and encouragement.

John Mellor-Crummey, Chuck Koelbel, Bradley Broom, Rob Fowler, Richard Hanson, and Mike Brogioli provided help and advice for which I am very appreciative.

I would like to thank my fellow students Tim Harvey, Qing Yi, and Arun Chauhan for their advice on this document. I would particularly like to thank Arun Chauhan for his tireless help throughout this project and for helping me have confidence in my ideas.

Lastly, I would like to thank my family, especially my husband Bem, for listening, helping, and putting up with my many mood swings during the completion of this work.

Contents

Abstract i

List of Illustrations vi

1 Introduction 1

1.1 Telescoping-Language Strategy 1

1.2 ARPACK as an Application of Telescoping Languages 3

2 ARPACK 6

2.1 Types Supported by ARPACK 6

2.2 Reverse Comunication Interface 8

3 The Type-Inferencing Problem 11

3.1 Features of Matlab 11

3.2 Definition of Type 13

3.3 Need For Forward and Backward Flow of Information 14

3.4 Whole-Program Analysis 15

3.5 Propositional Formulation 16

3.5.1 Example of Constraints for Size 17

3.5.2 Combining the Information 19

4 An Efficient Algorithm for Size Inference 21

4.1 Assumptions 21

4.2 Preliminary Analysis Required 23

4.2.1 Inferring the Number of Dimensions 23

4.2.2 SSA 23
4.3 Reducing to the Clique Problem 24
4.3.1 Using the Graph to Find a Whole-Procedure Solution 26
4.4 Description of the Algorithm for Straight-Line Code 27
4.4.1 Finding n-Cliques 27
4.4.2 Solving the Equations 32
4.5 Extending the Algorithm to the General Case 33
4.5.1 Variable Sizes not Determined by Input Parameters 33
4.5.2 Slicing 35
4.6 Some Remaining Issues 35
4.6.1 Handling Procedure Calls 36
4.6.2 User-Defined Annotations 36
4.6.3 Constants 37
4.6.4 Array Accesses 37
4.6.5 Recombining Arrays 38
4.7 The Result 39
4.8 Code Generation 39

5 Inferring Intrinsic Types and Shapes 41

5.1 Intrinsic Types 41
5.1.1 Forming Constraints 41
5.1.2 Solving the Equations 43
5.1.3 Handling Control Flow 43
5.2 Shape 43

6 Experimental Evaluation 45

6.1 Analysis 45
6.2 Experiments 46

7 Related Work **52**

7.1 Work in Translating Matlab 52

7.2 Type Inferencing in the Programming Languages Community 53

7.3 Constraints Logic Programming 54

8 Conclusion **55**

Bibliography **56**

Illustrations

1.1 Graphical Model of Telescoping-Language Approach 2

2.1 ArnoldiC -Matlab subroutine 7
2.2 Reverse Communication Interface 9

3.1 Matlab Type-Hierarchy 12
3.2 Example Where Backward Propagation is Useful 15

4.1 Example Graph 25
4.2 Iterative n-Clique finding algorithm. 29
4.3 Iterative n-Clique Finding Algorithm 30
4.4 Slice Hoisting 36
4.5 Handling Constants 37
4.6 Dealing with Subscripted Array Access 38

5.1 Intrinsic-Type Lattice 42

6.1 Pruned SSA Form of ArnoldiC 48
6.2 Constraints on Statements within the Loop 49
6.3 Configurations Satisfying Cliques 50
6.4 Comparison of Arnoldi Subroutines 51

Chapter 1

Introduction

High performance is critical for scientific applications. However, the man-hours required to attain high performance and the limited number of expert programmers inhibits the productivity of scientists. High-level scripting languages are much easier to manage, as is evidenced by the popularity of Matlab® in the scientific community, but incur a significant cost in performance. Telescoping languages refers to a strategy for numerical applications that promotes the use of high-level languages to increase productivity while maintaining performance comparable to that of lower-level languages.

The contribution of this thesis is twofold. First, it shows that the telescoping-language strategy can be used to generate code comparable to the Fortran version of ARPACK, a linear algebra library, from Matlab development code. Second, it provides an efficient algorithm for inferring types in Matlab to compute type jump functions on the library subroutines (needed to accomplish the former and to infer types interprocedurally). Through these contributions, I demonstrate the practicality and viability of Telescoping Languages and provide a basis for a general system for telescoping languages [15, 16].

1.1 Telescoping-Language Strategy

Traditionally, library compilers have not had knowledge of the possible calling contexts of the subroutines being compiled. This meant that compiling user scripts often involved recompiling the library subroutines to optimize the library based on the new

®Matlab is a registered trademark of MathWorks Inc.

information about the calling context. Therefore, compiling small scripts could take large amounts of time, since compile time was proportional to the size of call graph and not the script size. Telescoping languages avoids this problem by having the library writer annotate the subroutines with information about the possible calling contexts. Specialized variants of the subroutine are then generated by the library compiler, and the script compiler would determine which of the already-compiled subroutines to use based on the calling context. This allows the user script to treat library calls as primitive operations in a higher-level language. In fact, most operations in high-level languages translate to library calls. Therefore, the library compiler can be thought of as a language generator, hence the name Telescoping Languages. Figure 1.1 demonstrates this strategy graphically.

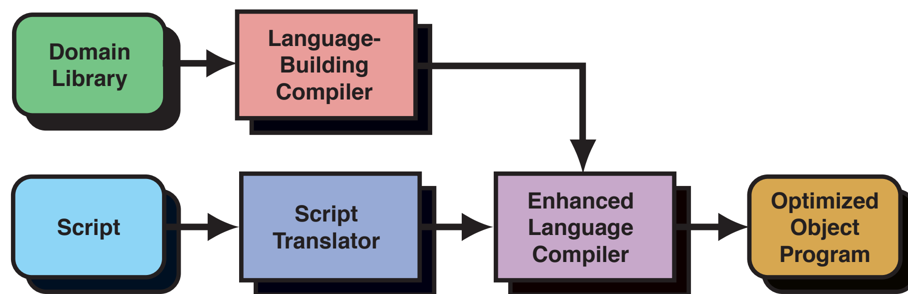

Figure 1.1 : Graphical Model of Telescoping-Language Approach

Library writers can also benefit from using telescoping-language technology in developing their code. Telescoping languages allows library writers to develop and maintain their code in a high-level scripting language and transfers the responsibility for achieving high performance to the library compiler. The library compiler must have an extensive analysis phase, after which it can optimize and specialize based on the information found. It generates Fortran or C code as an intermediate language so that it can leverage the already-existing, highly-tuned vendor compilers to aid in optimization.

To demonstrate how the library compiler works, I use Matlab as the high-level scripting language because of its popularity among scientists, including the writers of ARPACK.

1.2 ARPACK as an Application of Telescoping Languages

ARPACK is a library for numerical applications used to solve large-scale eigenvalue problems[25, 2]. Specifically, it implements a variant of the Arnoldi Process called the Implicitly Restarted Arnoldi Method (IRAM). ARPACK was initially designed in Matlab and then hand-translated into Fortran 77 for better performance. By starting in Matlab, the library designers were able to develop their algorithms in the simpler environment, thereby allowing them to have a better grasp of the implementation, behavior, and issues of the problem than if they had started in Fortran.

ARPACK is a useful and relatively simple example of an application that can benefit from telescoping-language technology. This thesis uses ARPACK to explore some ideas of telescoping languages in order to develop ways of achieving its goals. Specifically, this thesis explores the need for inferring types for translation into a lower level language, for specialization and optimization based on types, and for computation of type jump functions needed for interprocedural analysis.

The writers of the Fortran ARPACK found it useful to create specialized procedures based on types. The existence of these specialized variants indicates that it is important to distinguish between the different intrinsic types and array layouts. Inferring the size of the matrices can also lead to possible optimizations. For example, it can help avoid reallocating the arrays at every iteration of a loop, since Matlab arrays can grow.

For this thesis, I use ArnoldiC, a subroutine written in Matlab that implements the Arnoldi process, to illustrate translating an ARPACK subroutine from Matlab into Fortran. Due to the lack of defined types in Matlab, the compiler needs to be able to infer possible types for the variables in ArnoldiC.

The Matlab code is used to automatically produce code comparable in performance to the Fortran version of ARPACK, thereby allowing the developers to both bypass the translation process and maintain the library in the simpler environment. As the compiler translates, it optimizes the code and creates specialized variants based on possible input types and calling contexts. Since this is done automatically, the compiler can generate more specialized variants than those found in the Fortran ARPACK, since the number of specialized variants in ARPACK was limited by the time and energy of its creators. Experiments show that specializing for different types is important in the context of ARPACK.

In order to achieve all of the above-stated goals, I have developed and used telescoping-language technology to perform type-inferencing and specialization. To demonstrate the technology, this thesis develops a system, ARGen, that automatically produces an equivalent to the Fortran ARPACK from the Matlab development-code by developing and utilizing telescoping-language technology. By using this system, the library developers can avoid the step of hand-translating the Matlab code to Fortran, thereby greatly reducing library-development time. The writers also have the benefit of high performance, since the code generated by the system roughly achieves the performance of the hand-written Fortran code. Specialized variants are automatically generated to avoid long compilation of user scripts. The ideas and algorithms used to develop ARGen and described in this thesis can be used in constructing a general system in the telescoping languages construct that allows library writers to develop and maintain their code in high-level languages.

This thesis develops telescoping-language technology for automatically generating high-performance libraries from high-level development code. It generates optimized variants of the subroutines based on type and computes type jump functions on each of the subroutines to aid in analyzing calling subroutines. To accomplish all this, this thesis develops an efficient algorithm for inferring types in high-level languages. The contributions of this thesis are demonstrated using Matlab as the example high-level

language and ARPACK as the example library.

I first describe ARPACK in more detail in Chapter 2 to discuss those features that need to be handled by ARGen. In Chapter 3, I discuss the type-inferencing problem as it applies to Matlab and ARPACK and present a propositional formulation used for solving the problem. I demonstrate an efficient algorithm for inferring sizes in Chapter 4 using the propositional formulation. I then show, in Chapter 5, how this algorithm can be used in other type-inferencing problems. I apply the algorithm to a subroutine from the Matlab code and give experimental results of the generated code in Chapter 6. Chapter 7 describes related work. The contributions of this thesis are summarized in Chapter 8.

Chapter 2

ARPACK

ARPACK stands for ARnoldi PACKage. It is a collection of Fortran 77 subroutines designed to solve large-scale eigenvalue problems. ARPACK implements a variant of the Arnoldi Process called the Implicitly Restarted Arnoldi Method (IRAM). The most important aspect of ARPACK for the purpose of this thesis is that the code was designed using Matlab and then translated to Fortran 77 by hand.

ArnoldiC is a subroutine from a Matlab version of ARPACK. It corresponds to the subroutine XYaitr in the FORTRAN version. Figure 2.1 shows the code for ArnoldiC. I use this subroutine throughout this thesis as the operative example.

This thesis demonstrates the practicality and power of telescoping languages by generating code equivalent to that in the Fortran ARPACK from the Matlab code. Because Matlab is weakly typed, the primary analysis needed in ARGen is type inferencing. Because ARPACK supports several input types, ARGen should infer that these types are important and produce specialized variants for each.

2.1 Types Supported by ARPACK

ARPACK supports all types XY, where X can be:

single precision real arithmetic,
double precision real arithmetic,
single precision complex arithmetic, or
double precision complex arithmetic,

and Y can be:

$$
\begin{array}{l}
function[V,H,f] = ArnoldiC(A,k,v);\\
\quad v = v/norm(v);\\
\quad w = A*v;\\
\quad f = w - v*alpha;\\
\quad c = v'*f;\\
\quad f = f - v*c;\\
\quad alpha = alpha + c;\\
\quad V(:,1) = v;\\
\quad H(1,1) = alpha;\\
\quad forj = 2:k,\\
\qquad beta = norm(f);\\
\qquad v = f/beta;\\
\qquad H(j,j-1) = beta;\\
\qquad V(:,j) = v;\\
\qquad w = A*v;\\
\qquad h = V(:,1:j)'*w;\\
\qquad f = w - V(:,1:j)*h;\\
\qquad c = V(:,1:j)'*f;\\
\qquad f = f - V(:,1:j)*c;\\
\qquad h = h + c;\\
\qquad H(1:j,j) = h;\\
\quad end
\end{array}
$$

Figure 2.1 : ArnoldiC -Matlab subroutine

non-symmetric, or

symmetric.

There are specialized subroutines for each possible combination of X and Y. The XYaitr ARPACK subroutines correspond to ArnoldiC for the different types mentioned. The above types occur frequently and have the greatest need for specialization, either for optimization (as in symmetric vs. non-symmetric), or for correctness (complex vs. real). The developers of ARPACK saw the benefit and necessity of having specialized subroutines based on type.[1] This specialization fits within the telescoping languages framework and motivates this thesis. The compiler can automatically generate variants for even more specific cases (i.e., banded matrices) because it does not face the same time constraints as the ARPACK designers.

2.2 Reverse Comunication Interface

ARPACK uses a reverse communication interface, which allows the library users to further specialize the routine based on their particular needs. The reverse communication interface requires the user to provide the matrix-vector multiplication from the subroutine. In ArnoldiC, this operation is written as `w = A * v`. Since Fortran 77 does not allow functions to be passed as parameters to the routine, the interface is a necessary work-around to give users the ability to provide this routine.

The interface is shown in Figure 2.2. It calls for the control to jump out of the subroutine whenever the product is required. It then loops back to call the top level routine using the flag, `ido`, to indicate where it left off in the computation.

This interface allows users to specialize the matrix-vector multiplication based on certain properties of the input matrix `A`. In some instances, the users do not even have to provide the matrix. Rather, given the vector, they can provide a routine

[1] Specialization based on type is common to most Fortran libraries

```
10 continue call snaupd (ido, bmat, n, which,...,workd,..., info)
    if (ido .eq.  newprod) then
         call matvec ('A', n, workd(ipntr(1)), workd(ipntr(2)))
    else return
    endif
    go to 10
```

Figure 2.2 : Reverse Communication Interface

that computes the result of the multiplication given `v`. The reverse communication interface does not appear in the original Matlab code, since it is not necessary in Matlab.[2] Therefore, the interface does not appear in the generated version. Future research may account for the reverse communication interface as an option for the user, since the user may not want to provide a matrix or may need more refined specialization.

However, because the compiler is able to specialize for more types than the library writers had time to develop, it can handle some of the extra cases. In the example test code given by ARPACK, one of the special cases handled by the reverse communication interface required that matrix A be banded, even though there was not a version of the subroutine for banded matrices in the Fortran ARPACK. However, the automatically generated code could account for this case as a separate specialized variant if the library writers included annotations indicating that this case might be important.

Due to space constraints, the compiler cannot generate a specialized variant for every case a user of ARPACK might want, nor cannot it bypass forcing the user to represent the matrix explicitely and in a specific way. However, the reverse communi-

[2]The library development code should be written in the simplest form possible, and should use primitive operations where necessary, since it is the compiler's responsibility to find the most optimal routine for this operation in the lower-level language.

cation interface adds overhead that the automatically generated variants would not, making the automatically generated version is slightly more efficient for common or expected cases.

Chapter 3

The Type-Inferencing Problem

Telescoping languages proposes having an extensive analysis phase during library compilation. Part of this phase involves generating specialized variants of each library subroutine based on the possible calling contexts found through the analysis. When compiling the user script, the subroutine corresponding to the specific calling context can be used, avoiding the need for recompilation. The library compiler must infer all possible types for a subroutine to generate variants that can handle every possible calling context and to compute type jump functions for the subroutine. While type information is necessary to generate lower-level code from Matlab, it can also be used by optimizations that rely on accurate type information.

There are several issues involved in inferring types in Matlab. One complication is that operations are heavily overloaded. Also, types of variables can be inferred from their uses as well as their definitions, which makes both forward and backward flow analysis useful.

3.1 Features of Matlab

The simplicity of the Matlab syntax is well suited to increasing coding productivity. However, some of the very features that make Matlab ideal for development purposes are a hindrance when translating applications to a lower-level language such as FORTRAN or C. These features include:

- Matlab is weakly typed. This means that inferring types is necessary to translate Matlab code into a lower-level languages, all of which require declared types.

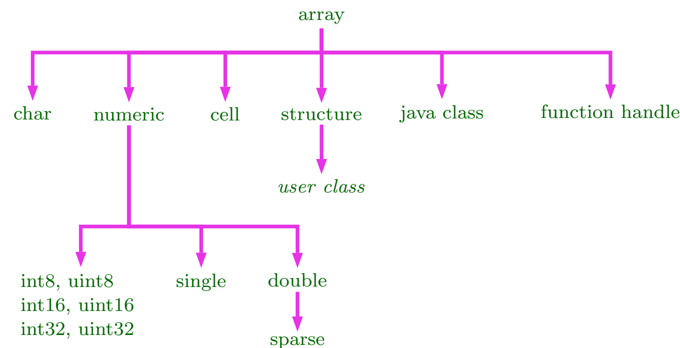

Figure 3.1 : Matlab Type-Hierarchy

- Matlab allows types to change if they are redefined in the middle of the subroutine. For example, arrays can grow, and each growth would require reallocating the array. This severely hampers performance when an array grows within a loop (which happens in ArnoldiC).
- As can be seen from Figure 3.1, Matlab treats all variables first and foremost as arrays, including scalars, which are represented as 1×1 arrays.
- Operations in Matlab are heavily overloaded. For example, in the statement `w=A*v`, the `*` operation could be interpreted as matrix multiply if both `A` and `v` are matrices. However, if `A` is a scalar, then the operation should be interpreted as multiplying every element of `v` by `A`. The operation used affects the size of w, which, in turn, affects other sizes that are defined using w.

Because of these features, the compiler often cannot statically determine an exact type for every variable. However, in some cases the library writer may have intended multiple interpretations of the same code. The overloaded operators are, in fact, why Matlab is a much simpler language for developing code, because one subroutine can handle many different kinds of input. The compiler must account for each of the intended possibilities while limiting the number as much as possible to avoid having to generate more specialized variants than are necessary.

ArnoldiC, a Matlab ARPACK subroutine, was intended for both complex and real arrays. In converting ArnoldiC into Fortran, ARGen needs to generate code for each possibility, not only because Fortran requires a variable's type to be declared, but also because for correctness and efficiency different operations must be used depending on whether the matrix is complex or real. The compiler has to provide separate code for each case as well as code that determines types at runtime that could not be determined statically.

3.2 Definition of Type

The kind of type information needed by the compiler depends both on what the scripting language (Matlab) allows and also what the destination language (Fortran) needs. Because Matlab treats all variables as arrays, the compiler needs to use an extended notion of type (based on De Rose's work) [9, 10].

A variable's type, for the purposes of this thesis, is defined as a tuple $<\tau, \rho, \sigma, \psi>$ where,

- τ refers to intrinsic type such as int, real, complex or char (needed by the compiler to declare a variable in Fortran),
- ρ refers to an upper bound on the number of dimensions of the variable (needed to allocate space in Fortran),
- σ is a tuple showing the size of the array in each of the ρ possible dimensions (also needed for memory allocation in Fortran), and
- ψ is the shape of the array, such as sparse, banded, etc. (useful for optimization).

Knowledge of these types is important for both code generation and specialization/optimization. In this thesis, I initially concentrate on using the algorithms to infer size, and I will describe in Chapter 5 how the ideas and algorithms can be applied to determining intrinsic type and shape.

3.3 Need For Forward and Backward Flow of Information

Type information flows in two directions[9, 10]. The variable's type information flows forward from its definition to its uses and also backwards from its uses to its definition. To ensure that each use is reached by only one definition, the compiler must convert the code to static single assignment form, or SSA, before analyzing the code. SSA is a standard compiler technique to eliminate artificial sharing of names among unrelated values. Redefining part of an array is treated as a redefinition of the entire array (i.e., the array is renamed at this point), since the redefinition could cause a change in type for the original array in addition to a change in value. These arrays can be merged together after analysis if only a limited change in type was inferred. I will discuss merging arrays more fully in subsequent chapters.

Information about the type of a variable can be determined at the definition point based on the operation and the inputs involved. This type information can then be used at a later point when another variable is defined using the previously defined variable.

If the variable's exact type could not be determined at its definition, information about its type can also be determined by the variable's uses. As one example, if the variable is an input to an operation that accepts only a known type, then it can be assumed, given correct code, that the variable is of that type. In Matlab, information about variable types flows both forward and backward.

Figure 3.2 shows a piece of code from the Matlab ArnoldiC subroutine, which demonstrates an example of where backward flow of information is useful. The size of `v` is not clear from the definition, but line 7 explicitly refers to `v` as a vector. This information can lead to better information about variables defined by and used with `v`. By taking into account information flowing in both directions, it is possible to find the strictest possible types for the variables.

```
1    v = v/norm(v);
2    w = A*v;
3    f = w - v*alpha;
4    c = v'*f;
5    f = f - v*c;
6    alpha = alpha + c;
7    V(:,1) = v;
8    H(1,1) = alpha;
```

Figure 3.2 : Example Where Backward Propagation is Useful

3.4 Whole-Program Analysis

There are two difficulties to using data flow analysis for solving the type-inferencing problem:

1. It is difficult, if not impossible, to determine if the analysis halts on a given subroutine. This is due, in part, to the fact that information flows in both directions. Also, the lattices involved in solving the problem do not meet all the requirements for proving termination, namely a finite lattice and a monotonic meet operation (i.e. the result of the meet operation could be above or below its arguments on the lattice). When inferring sizes, the lattice has infinite depth. When inferring intrinsic types and shapes, the meet operations on the lattices are not monotonic.

2. The compiler needs to find all of the solutions allowed by the problem. Therefore, data-flow-analysis is ill-suited for the problem, since if the analysis converges, it converges to a single solution or to a single general set of types. In other words, it would not be able to compute the relationships between the possible variable types. In order to solve the problem using conventional data-flow analysis, the compiler would have to run several passes of analysis under all possible assumptions about the variable types. This is extremely cumbersome.

I propose a different solution. Rather than performing forward and backward analysis, the compiler can gather the information by simultaneously determining the possible types of the variables involved in the subroutine. The compiler does this by looking at the restrictions each individual operation or procedure call[1] places on the variables involved in it, and from these, forming sets of legal configurations of the types of all the variables in the operation or procedure call. It then uses the intersection of the statement-by-statement sets to determine the possible configuration over the whole program. The expected result from whole-program analysis is a list of possible type-configurations, each of which cover every variable in the subroutine.

3.5 Propositional Formulation

I chose propositional logic to represent the constraints on the types of the variables because it is powerful enough to represent the relationship between variables over a subroutine yet simple enough to use for practical purposes. Because the code is in SSA form, each statement gives information about the variables that must be true over the entire program. The statement constraints are formulated using boolean expressions, and the conjunction of these statements represents the constraints over the entire subroutine.

In forming the statement constraints, the compiler first enumerates each possible configuration of types for the variables involved in the statement. There should be fewer than t^v configuration, where t is the number of possible type-values for any variable and v is the number of variables involved in the statement. It then takes the disjunction of all of these possibilities as the constraint for the statement. Each constraint lists the possible type configurations for the variables.

I demonstrate the constraints using the size-inference problem.

---

[1]Operations and procedure calls are handled in the the same way, so I will use them interchangeably. Also, I will often refer to both of these as statements and assume statements have only one operation or procedure call.

3.5.1 Example of Constraints for Size

First, I need two kinds of information from the constraints.

1. I need to know whether the variables are scalars or arrays. This determines which operation should be used.

2. Since the sizes of all the variables are not known at library compilation time, I want to know the size of each variable in terms of the sizes of other variables in the subroutine, so that when additional information about the variables is known (i.e., the sizes of the inputs), the sizes of the remaining variables can be easily inferred. This is important for allocation of arrays.

The following statement, taken from ArnoldiC, demonstrates how statement constraints are represented for the size inferencing problem (a maximum rank of 2 is assumed for all variables).

$w = A * v$

would use the annotation from a database for $vout_1 = vin_2 * vin_3$, which is:

$(\sigma^{vin_1} = <1, 1> \ \& \ \sigma^{vin_2} = <1, 1> \ \& \ \sigma^{vout_1} = <1, 1>) \|$

$(\sigma^{vin_1} = <1, 1> \ \& \ \sigma^{vin_2} = <\#1, \#2> \ \& \ \sigma^{vout_1} = <\#1, \#2>) \|$

$(\sigma^{vin_1} = <\#1, \#2> \ \& \ \sigma^{vin_2} = <1, 1> \ \& \ \sigma^{vout_1} = <\#1, \#2>) \|$

$(\sigma^{vin_1} = <\#1, \#2> \ \& \ \sigma^{vin_2} = <\#2, \#3> \ \& \ \sigma^{vout_1} = <\#1, \#3>) \|$

$(\sigma^{vin_1} = <1, \#1> \ \& \ \sigma^{vin_2} = <\#1, 1> \ \& \ \sigma^{vout_1} = <1, 1>)$

which would translate to constraints:[2]

[2] The first clause corresponds to scalar multiplication since all the sizes in all the dimensions for all the variables are 1. The fourth clause corresponds to a matrix-matrix multiplication, since not all dimensions for the variables are 1.

$(\sigma^A = <1,1> \; \& \; \sigma^v = <1,1> \; \& \; \sigma^w = <1,1>) \|$

$(\sigma^A = <1,1> \; \& \; \sigma^v = <\$1,\$2> \; \& \; \sigma^w = <\$1,\$2>) \|$

$(\sigma^A = <\$1,\$2> \; \& \; \sigma^v = <1,1> \; \& \; \sigma^w = <\$1,\$2>) \|$

$(\sigma^A = <\$1,\$2> \; \& \; \sigma^v = <\$2,\$3> \; \& \; \sigma^w = <\$1,\$3>) \|$

$(\sigma^A = <1,\$1> \; \& \; \sigma^v = <\$1,1> \; \& \; \sigma^w = <1,1>)$

if $1 had not been used yet for any other statement in Arnoldi.

The annotations and constraints define each dimension of σ for each variable in terms of the other dimensions of the variables by using the $ and # variables as place-holders.

The compiler would be able to obtain the annotation from a database. In forming the constraints it would replace the annotated arguments with the actual argument names, and the # variables with $ variables that have not yet been used in forming the constraints over the previous statements. No constraints on any two distinct statements can involve the same $ variable because each $ variable is only a place-holder for the variables in a single operation or procedure call, and these statement constraints are combined to form the whole-subroutine constraint. If a $ variable could be used for more than one statement constraint, the conjunction of the statement constraints would be meaningless, since unrelated variables could have dimensions defined with the same $ variable.

Each clause in the constraint is necessary, since Matlab handles scalars differently from arrays. The clauses must be mutually exclusive in order to prove properties about the constraints, which will be necessary in finding an efficient algorithm. The $ variable is a dummy variable, used to show the relationship between the sizes of the variables in the statement. If the same $ variable is used for two different variable dimensions, it is assumed that those variables have the same size in those dimensions.

To distinguish scalars from arrays, the presence of a dimensions size that is not 1 means that the variable is not scalar. Therefore, if the size constraint for both

dimensions of a variable is represented by $ variables, it is assumed that it cannot be scalar (both dimensions cannot be of size 1). However, since Matlab treats vector operations similarly to matrix operations, only one of the dimensions need not be 1.

To sum up, for size-inferencing there are two possibilities for each variable. Either the size of each dimension is 1 (i.e. the variable is scalar), or the size of each dimension is based on the sizes of the dimensions of the other variables in the statement. Vectors are only special cases of matrices, and as such do not need a separate case. Null vectors and matrices are handled the same way as regular non-scalars, since they behave the same way as non-scalars. Null vectors will have a dimension of size 0. These constraints satisfy both of the requirements for size inferencing needed by ARGen. Scalars and arrays are distinguished and $ variables fulfill the function of place-holders representing how the sizes of the different dimensions of all the variables are related.

Claim 3.5.1 *There are a finite number of clauses for each statement constraint.*

Proof: Since there are only two cases for each variable (scalar or otherwise), and since I assume the number of parameters involved in each statement is bounded by a small constant v,[3] there should be only 2^v possible clauses. □

In practice, there are often fewer clauses, since many combinations might be illegal for the operation. For example, multiplying two scalars will never produce a matrix.

3.5.2 Combining the Information

The constraint over the whole subroutine is the intersection of all the statement constraints. The intersection corresponds to taking the conjunction of all the constraints and finding all possible type configurations for all the variables in the program that

[3]This is a reasonable assumption since the number of variables involved in an operation or procedure call does not grow with the size of the program[7].

satisfy the conjunction. The SAT problem can easily be reduced to this problem, showing that it is NP-hard in the general case. However, the next chapter will describe an algorithm for finding this intersection, which takes advantage of certain properties of this problem to reduce the complexity.

Chapter 4

An Efficient Algorithm for Size Inference

The constraints over the entire subroutine are found by taking the conjunction of the statement-constraints (discussed in the previous chapter) and finding all type configurations that satisfy the resulting boolean equation. This chapter presents an efficient algorithm for doing this using the specific properties of the problem.

To describe the algorithm, I will use size inference as an example. I will then describe how to modify the algorithm to infer the other types in the next chapter.

4.1 Assumptions

There are a number of assumptions necessary for this algorithm to perform correctly.

1. The algorithm assumes that it has a correct program on input.[1] Although in some cases the compiler may be able to determine that a program is incorrect (i.e., when it proves that the whole-subroutine constraints cannot be satisfied), for the most part, proving correctness is not the responsibility of the algorithm. The algorithm relies on the assumption of correctness in making its decisions. This is one of the primary differences between this type-inferencing algorithm and others from the programming languages community.

2. Although Matlab allows passing functions as parameters by passing the name of the function as a string, the algorithm will assume that this feature is not

[1] This is a reasonable assumption for Matlab programs since users can develop and test their code in the Matlab interpreter before giving them to the optimizing compiler.

used, since it assumes all input parameters are array values. Accounting for this feature is left for future research.

3. The algorithm requires that the number of input and output parameters in each operation or procedure is less than m, where m is a small constant. This is important for the complexity of the algorithm to remain small. This is a reasonable assumption, since parameter lists do not tend to grow with the size of the function.[7]

4. The algorithm also assumes that there are no global variables. The library writer can avoid this problem by making those variables an input and an output to every procedure call. Since the compiler assumes that the number of parameters is bounded by a constant, the number of variables used in this way must be limited as well. The compiler could also automatically convert global variables. Well-written libraries rarely use global variables, so again this is not a major obstacle to using the algorithm described.

5. The algorithm requires the compiler to already have annotations, described in the previous chapter, on all the operations and subroutine calls in the procedure. The compiler keeps a database of these annotations that describe the different possible input and output parameter type configurations for all the operations or subroutine calls. These annotations are either entered by hand, for primitive operations, or, by the compiler after analysis for analyzed subroutines. The annotations for user-defined procedures will look the same as the hand-entered annotations since they are handled exactly as primitive operations by the algorithm. Generating these entries will be described later.[2]

---

[2]Recursive calls invalidate this assumption since the compiler will not have done the analysis on the subroutine prior to the call. Recursive calls will also be dealt with later in this section.

4.2 Preliminary Analysis Required

In order to infer sizes over the entire program, the compiler must first perform some preliminary analysis. It has to know for which dimensions it needs to infer size. Knowing the maximum number of dimensions would give the compiler this information. It also needs to put the code in SSA form, so that each statement constraint on the variables will hold over the entire procedure, since SSA ensures that each variable has a single value and therefore, a single type in straight-line code.[3]

4.2.1 Inferring the Number of Dimensions

Recall from Chapter 3 that σ is a tuple consisting of ρ fields, where each field is the size of the variable in the corresponding dimension and ρ is an upper bound on the number of dimensions. In order to infer σ, the compiler needs to first determine ρ. The compiler only needs ρ to be an upper bound, since size-inferencing will be able to tighten the number of dimensions by inferring that certain dimensions have size 1.

To get the ρ information, the compiler must perform a single prepass over the code to see which dimensions are accessed in which variables, either by direct-subscripted accesses or by operations. Some operations also require that the number of dimensions for the variable be limited.

When the prepass cannot determine a bound, the compiler must create a dummy dimension field in σ for the variable representing all dimensions that may behave differently from the rest. This handling of the extra dimensions is valid since they are not accessed explicitly. Therefore, they must have determinable behavior.

4.2.2 SSA

The algorithm also assumes the code is in SSA form. That is, every use of a variable is reached by exactly one definition. Since redefining a variable could change the

[3] ϕ nodes, used in SSA to handle join points in the control flow graph also help the algorithm when dealing with control flow.

variable's type, the compiler would have difficulty reasoning about this variable's type over the entire program without this property. Also, redefining variables to be different sizes would make allocation of the variable difficult in Fortran.

Since variable sizes could grow if elements outside the original array are accessed on the left-hand side of some statement, the compiler will consider this to be a redefinition of the entire array variable (as is consistent with the traditional notion of SSA). The compiler will have to emit code to copy the other values of the array. This is necessary for the algorithm to work correctly since it will assume that variable sizes do not change. Also, since Fortran requires that the variables have static sizes, and since the previous references to the variable may require it to be scalar, this is necessary for correct translation into Fortran. If the variable does not grow from a scalar to an array, these separated variables can be merged into an array of the maximum size in the code generation phase.

Using SSA form will also be helpful when dealing with control flow, which will be discussed later in this chapter.

4.3 Reducing to the Clique Problem

Once the possible constraints for each statement have been determined as described in the previous chapter, the compiler needs to analyze them over the whole program. That is, it needs to find all possible configurations of sizes for the variables that satisfy the overall constraints. By representing the statement constraints as nodes in a graph, the problem is reduced to that of finding n-cliques, where n is the number of statements in the program.

Constructing the Graph

Figure 4.1 shows how the graph is constructed. Each possible size configuration for that statement or clause is represented by a node at the level that corresponds to its statement number. There is an edge from one node to another if the equations

$A = b + c$

$1a$	$(\sigma^A =<1,1> \ \& \ \sigma^b =<1,1> \ \& \ \sigma^c =<1,1>)\vert$
$1b$	$(\sigma^A =<\$1,\$2> \ \& \ \sigma^b =<1,1> \ \& \ \sigma^c =<\$1,\$2>)\vert$
$1c$	$(\sigma^A =<\$1,\$2> \ \& \ \sigma^b =<\$1,\$2> \ \& \ \sigma^c =<1,1>)\vert$
$1d$	$(\sigma^A =<\$1,\$2> \ \& \ \sigma^b =<\$1,\$2> \ \& \ \sigma^c =<\$1,\$2>)\vert$

$E = c - d$

$2a$	$(\sigma^E =<1,1> \ \& \ \sigma^c =<1,1> \ \& \ \sigma^d =<1,1>)\vert$
$2b$	$(\sigma^E =<\$3,\$4> \ \& \ \sigma^c =<1,1> \ \& \ \sigma^d =<\$3,\$4>)\vert$
$2c$	$(\sigma^E =<\$3,\$4> \ \& \ \sigma^c =<\$3,\$4> \ \& \ \sigma^d =<1,1>)\vert$
$2d$	$(\sigma^E =<\$3,\$4> \ \& \ \sigma^c =<\$3,\$4> \ \& \ \sigma^d =<\$3,\$4>)\vert$

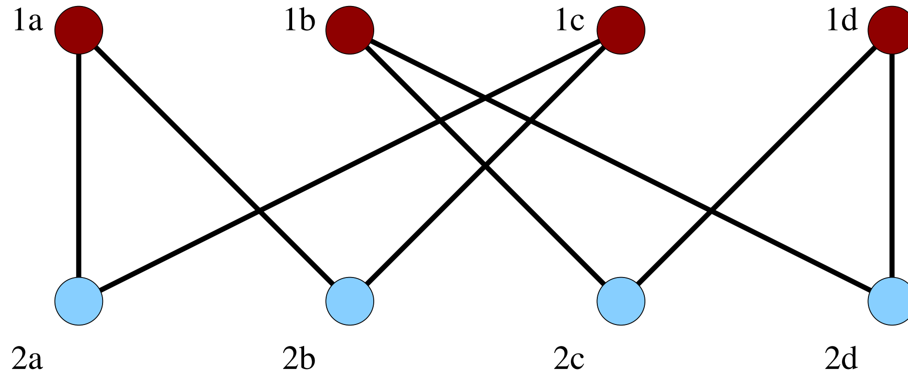

Figure 4.1 : Example Graph

in each node do not contradict one another. The only variable that appears in both statements in the figure is c; therefore, c determines if there is an edge from a node in one level to the next. Without the presence of c, the graph would be complete (i.e. there would be an edge from every node to every other node in the graph). There is an edge from node $1a$ to node $2a$ since c is scalar in both clauses. However, c is not scalar in clause $2c$ so there is not an edge from node $1a$ to node $2c$.

The final graph has n levels, where n is the number of operations or procedure calls. Each level has, at most, k nodes, where k is 2^v and v is the number of variables

involved in the statement. 2^v is the number of possible size configurations for each variable in the operation corresponding to that level, since there are two possible types for each variable (scalar or array/vector). Since by definition, the clauses are mutually exclusive, there will not be edges between nodes on the same level.

Complexity of building the graph

Building this graph takes $O(n^2)$ time. For each node at level i, the compiler has to compare the node with all other nodes at levels less than i. This must be done from every level i, so there are $k * \sum_{i=1}^{n}[k * (i-1)]$ comparisons, which restated is $\frac{1}{2}k^2 n * (n-1)$ comparisons. The comparisons require checking that a variable is involved in both nodes, and, if so, determining whether the right hand sides of the equations in which they are involved conflict (i.e., one right hand side shows that the variable is scalar while the other shows that it is multi-dimensional). Therefore, because the clauses have constant size, the comparisons only take a small, constant amount of time.

4.3.1 Using the Graph to Find a Whole-Procedure Solution

Now that the compiler has all the information in graph format, the problem is reduced to that of finding cliques in the constraint graph.

Finding possible constraints over the entire subroutine corresponds to finding sets of constraints, one from each operation or procedure call, that do not conflict with each other. Having at least one constraint from each operation or procedure call is necessary because otherwise the set of constraints would not hold over the entire subroutine. On the graph this would correspond to finding sets of nodes such that every node has an edge to every other node in the set, and there is one node from each level in the set. In fact, there can only be one node from each set since there are no edges connecting nodes on the same level. This is exactly the problem of finding n-cliques (where n is the number of levels in the graph) such that each cliques has one

and only one node from each level (these conditions are trivially met because there are no edges between nodes on the same level).

4.4 Description of the Algorithm for Straight-Line Code

To describe the basic algorithm, I start with the simplest case and assume the compiler is only analyzing straight-line code where the size of each variable in each statement can be determined by the sizes of the other variables in the statement (i.e. there are no data dependences). In subsequent sections, I will expand the algorithm to handle the general case.

4.4.1 Finding n-Cliques

The problem has been reduced to that of finding n-cliques in the graph where there is a node from each of the n levels.[4] Although finding n-cliques is an NP-complete problem, I show that the structure of the graph and the specific problem limit the complexity.[8]

Total Number of Possible Type-Configurations

Since the compiler is going to solve the equations in the cliques separately, I need to show that the number of cliques is manageable. Also, since the compiler will generate a variant for each clique, a large number of possibilities would cause a blow-up in the size of the generated code.

Claim 4.4.1 *In the absence of control flow with all variables defined in terms of other variables, the number of possible configurations of sizes is bounded by* 2^p*, where p is the number of input parameters.*

[4] A clique is a complete subgraph. A n-clique is a complete subgraph with n nodes.

Proof: Since the sizes of all the variables are determined by the sizes of other variables, all the sizes should ultimately depend on the sizes of the inputs. If this were not true, the behavior of the program would be indeterminable even at runtime, since variables not defined in terms of input would have no added information to determine their size at runtime. Therefore, the total number of possible configurations over all the variables will just be the number of possible configurations of the input parameters. Otherwise, the same set of inputs could produce different variations of types on the other variables. This contradicts the assumption that the sizes of all the variables are determined by the size of the inputs. □

What remains to be proven is that in the absence of control flow, the number of cliques is also bounded by 2^p.

Total Number of n-Cliques

To show that the number of cliques is bounded, I must show that no two cliques can reduce to the same set of equations once solved. If this is true, then it follows that each clique represents a different size assignment to the variables, of which, there can only be 2^p.

Claim 4.4.2 *The number of n-cliques is bounded by 2^p.*

Proof By Contradiction: I start by assuming there are two distinct cliques that represent the same size assignment to the variables. Then, at least at one level, to be distinct, the cliques must have "chosen" two different nodes. Since the statements in nodes of the same level contradict each other, at least one variable in the statement corresponding to that level must have a distinct size depending on which clique represents the size assignment used at runtime. Therefore, the two cliques cannot have the same size assignment to the variables. □

```
input:  graph G
output:  CurrCliques
initialize CurrCliques to be nodes on first level
1 for every level r in G after first
2   newCliques = empty
3   for every node n in r
4     for every clique c in CurrCliques
5     candidate = true
6       for every node q in c
7         candidate = candidate & edge?(n,q)
8       end for
9       if (candidate)
10        then newCliques = newCliques + clique(c, n)
11    end for
12  end for
13  CurrCliques = newCliques
14 end for
```

Figure 4.2 : Iterative n-Clique finding algorithm.

Iterative Algorithm for Finding n-cliques

Finding n-cliques is NP-complete. However, I claim that given the structure of the problem, there is an algorithm that will find n-cliques in polynomial time.

The solution is to build the cliques iteratively. Figure 4.2 shows the pseudocode for the iterative algorithm. The algorithm starts with one level and puts each node in that level in its own clique. It then compares each node with each already-formed clique. If the node has an edge to every member of the clique, it forms a new clique with the old clique. It does this for all levels of the graph after the first.

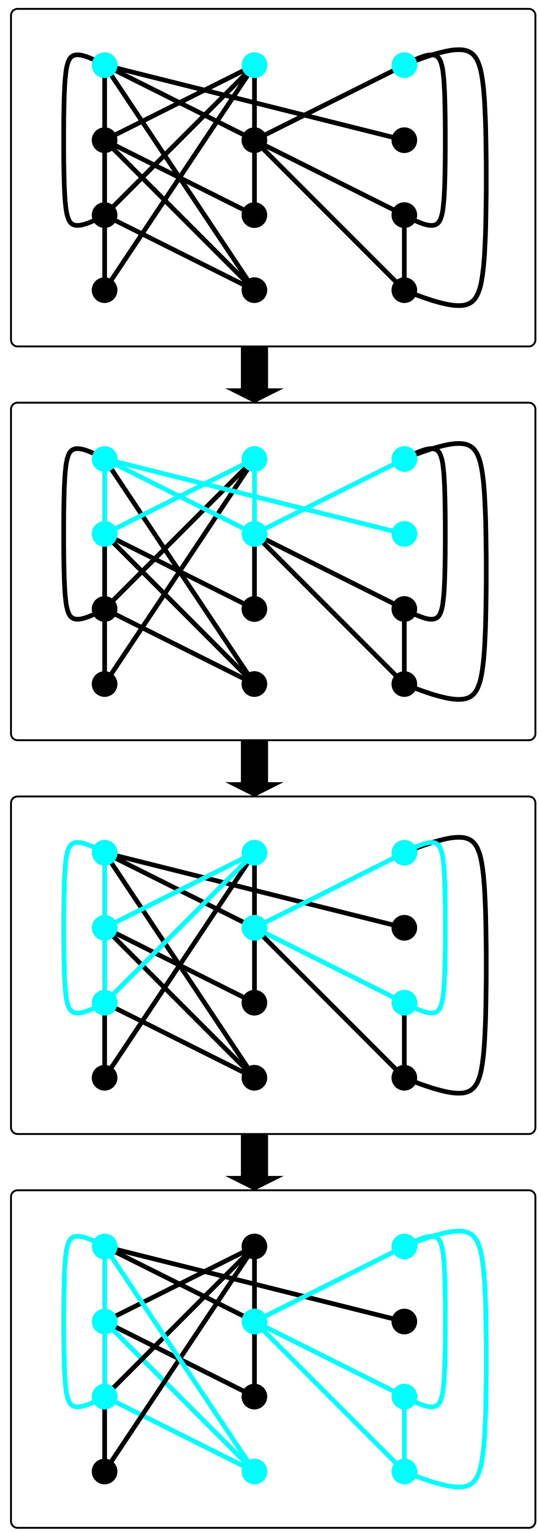

Figure 4.3 : Iterative n-Clique Finding Algorithm

Figure 4.3 demonstrates the algorithm on the given graph. In each step, the lighter nodes and edges are part of one or more cliques. At the first step, only the nodes in the first level are considered, and, of course, these nodes form cliques with themselves. At the second step, each node on the second level forms a clique with a node on the first level if there is an edge between them. The nodes on the third level look at all the cliques from the second step. If a node in the third level has an edge to both nodes in a clique from the second step, it forms a new clique. In the final step, there are only two cliques.

The complexity of this algorithm is still exponential in the worst case (as it should be given that finding n-cliques is NP-Complete), since line 4 in figure 4.2 could iterate over an exponential number of cliques from the previous step. With a limit on the number of cliques at each step, the complexity is also limited. Because the bound of 2^p only holds for the final number of cliques, I need to find a bound on the number of intermediate cliques to attain this limit. I therefore use the structure of the problem to prove a bound of 2^p at all intermediate steps if the levels are visited in program order.

Claim 4.4.3 *Claim: For the above algorithm, the number of cliques at any one step of the clique-finding algorithm is bound by* 2^p *if the order of the levels is the program order.*

Proof: We have from above that, given assumptions about how variable sizes are defined, on valid procedures (i.e. procedures where all variables are defined before they are used), there is an upper bound of 2^p on the number of cliques. In the absence of control flow, at any operation or procedure call the rest of the code can be left off, and the remaining (beginning) code is still valid, if meaningless. Therefore, since the algorithm is iterative and only finds cliques over the levels it has already seen, if the algorithm goes in the order of the operations, after every iteration the algorithm will have produced cliques on valid code. Therefore, the number of cliques after every

iteration must be bounded by 2^p. □

With 2^p cliques after every iteration the algorithm takes $k2^pn^2$ steps where n is the number of statements, k is the maximum number of nodes in a level, and p is the number of input and output variables per statement. Therefore, the overall complexity of the algorithm is $O(n^2)$.

Each of the cliques represents a type configuration for which a specialized variant based on those types may be formed. However, the types of the variables satisfying each clique have still not been determined. The equations in each clique need to be solved in terms of the input parameters to find the type allocations.

4.4.2 Solving the Equations

The compiler uses simple substitution to solve the constraints in each resulting clique. It should solve the equations in terms of the input parameters so that at runtime, the sizes of all the variables will be known when the input parameters are known. However, it may be possible for the compiler to infer exact types for some or all of the inputs in terms of the other inputs as well. This could present optimization opportunuties.

To handle cases where input types can be inferred, the compiler starts out by assuming it will be able to infer all possible sizes including the input parameters. If the solution does not converge, it will try to solve the sizes in terms of each parameter, and then each pair of parameters, etc. until it finds a minimal set of parameters that need to be held constant for the solution to converge. While this looks like a long process, this does not change the complexity of the algorithm because there is a constant bound on the number of input paramters.

The solver is really an independent piece. The current solver implementation is naive, but a more sophisticated solver can easily replace it without affecting the algorithm.

4.5 Extending the Algorithm to the General Case

The algorithm works well for straight-line code with no input or output operations and variable sizes being defined only by other variable sizes. However, code satsisfying these restrictions rarely occurs in practice. In order for the algorithm to be useful in practical applications, including ARPACK, it must be extended to handle the general case.

4.5.1 Variable Sizes not Determined by Input Parameters

The compiler may come across variables whose types do not depend on the types of the input parameters, but rather on which branches are taken within the procedure. In SSA form, these variables are always defined by ϕ nodes. ϕ nodes represent the points at which a variable may have multiple possible values depending on the control flow. This means that the variable could have multiple possible types as well, since any redefinition of the variable in Matlab could potentially change the type.

If the ϕ variable is used later, some other variables sizes may depend on the outcome of the control flow. Variables defined by ϕ nodes behave exactly as input variables, except that their sizes may not be known until the middle of the subroutine.

For now, the compiler does not consider the ϕ node to be an operation, and tries to solve the outcome of the variable defined by the ϕ node only from its uses. This works well for ArnoldiC, but future applications might benefit from treating the ϕ node as an operation and constraining the variable defined by the ϕ to be the types of the arguments to the ϕ node. This would only entail a minor change to the implementation, but could increase the number of levels in the graph by the number of ϕ nodes.

While the algorithm does not need to be changed to accomodate ϕ nodes beyond treating variables defined by ϕ nodes as inputs when solving the constraints, control flow complicates the algorithm in the following ways.

1. Having ϕ nodes increases the complexity of the algorithm since the number

of possible final cliques becomes 2^{p+c}, where c is the number of ϕ nodes in pruned SSA.[5] The number of intermediate cliques is also 2^{p+c} since ϕ nodes are always placed so that they dominate every node in the basic block. Therefore, the code will still be correct up to every point, since all variables are defined before they are used (except as arguments to the ϕ nodes). In order for the algorithm to remain polynomial, c cannot grow with the size of the program.[6] However, even if it does, the algorithm will still work correctly, though it may become exponential in the worst case. This case should not occur frequently in practice and does not occur in ArnoldiC. Actually, 2^{p+c} is an upper bound on the number of cliques, and this bound will not be reached if the subsequent uses of the variable defined by the ϕ node prove that the variable has a single possible type. ArnoldiC has this property. Most control flow is set up this way, since programmers are used to programming in lower-level languages which often require this property. Also, library subroutines in Matlab are typically small (in ARPACK under fifty lines of code).

2. The algorithm cannot create a specialized variant of the subroutine depending on the possible values of the ϕ nodes, since, when the subroutine is called the outcome of the ϕ node will be unknown, making it impossible to determine which variant to call. However, the compiler can create specialized paths within the subroutine based on the outcome of the ϕ nodes.

Variables defined by input statements can also be treated as input parameters. Again, the compiler cannot generate specialized variants of the subroutine, but can have specialized paths depending on the size of the inputs. The complexity is now increased to 2^{p+c+i}, where i is the number of input statements. For most scientific applications i should be small if not 0.

[5] Actually putting the code in pruned SSA form is not necessary since the algorithm ignores the outcomes of ϕ nodes if not used.

[6] The compiler assumes that code is in pruned SSA form.

Another issue occurs when variable sizes are determined by values of program variables that cannot be determined at library compilation time and variables that are defined by operations that could have multiple outputs for the same inputs. The occurance of these cases means that the compiler may not be able to statically determine whether the array is a scalar, unless it can determine that the value is never 1 or the output is never a scalar even for straight-line code. This causes the number of possible cliques to be 2^{p+c+d}, where d is the number of variables defined in this manner. Again, the addition of such variables will not cause a problem if bounded by a constant. The algorithm will still perform correctly even if d gets large.

4.5.2 Slicing

To handle determining sizes that cannot be inferred statically due to the issues mentioned in the previous subsection, the compiler uses an idea developed by my colleague Arun Chauhan, called slice hoisting.[4] The idea here is to make early runtime decisions by moving all the information relavent to the decision up to the earliest possible point. In the case of the ϕ node, it would move everything necessary for determining the outcome of the ϕ node to early in the program. This would mean moving the control flow and anything on which the control flow depends. Figure 4.4 shows how slicing works in the presence of a ϕ node whose size cannot be determined by the algorithm alone. Slice hoisting involves inserting σ statements that define the sizes of the variables that cannot be determined by the sizes of the inputs. Then the slice of the code that is involved in determining these σ statements is identified and hoisted before the array is allocated. Slice hoisting is performed after arrays are merged back together.

4.6 Some Remaining Issues

Several additional issues must be resolved for the algorithm to be useful in real applications, including ARPACK.

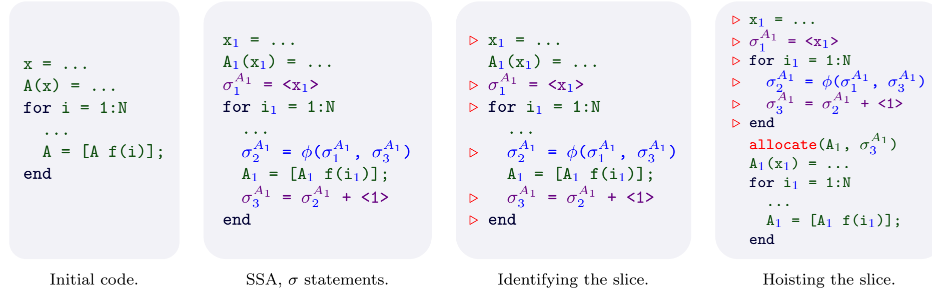

Figure 4.4 : Slice Hoisting

4.6.1 Handling Procedure Calls

The algorithm assumes that any called procedure has already been analyzed. If it encounters one for which the source code is not available and which, therefore, has no annotations, the algorithm simply ignores the call since it does not give any added information. This will degrade the analysis of the algorithm, although it does not affect the correctness. In essence, it says that the variables are unconstrained by the statement.

Mutual and Self-Recursion

Because recursive calls cannot have been analized prior to analyzing the calling subroutine, constraints cannot be generated at recursive call sites. For now, the compiler treats recursive calls just as it would unanalyzed procedure calls. While following the type information from the recursion to a fixed point could lead to more exact information, I leave this idea to be explored in future research.

4.6.2 User-Defined Annotations

One of the key concepts in telescoping languages is allowing the compiler to utilize the knowledge of the library writer through annotations. This information is important in that it could tell the compiler which cases can or cannot occur in practice, which

is something the compiler alone may not be able to infer.

For now, I assume the user-defined annotations concerning types are in the same form as the compiler-generated constraints. The compiler treats the user-defined constraints as constraints on the subroutine header, which corresponds to the zeroeth level in the graph. Any cliques occuring in the graph are forced to have part of the user-defined annotations as one of their nodes. This can greatly reduce the number of possible specialized variants the compiler would have to generate, and in the case of shape, could allow for finer optimization.

4.6.3 Constants

If one of the input arguments to an operation/procedure involves a constant, the compiler can just substitute the known information about the constant into the operation's annotations on the other variables. This should reduce the number of nodes in the graph. Figure 4.5 shows an example of how constants will be handled.

$w = A * [1\ 2\ 3\ 4]'$
would now show constraints:

$(\sigma^A =< 1, 1 > \ \&\ \sigma^w =< 4, 1 >)\|$
$(\sigma^A =< \$1, 4 > \ \&\ \sigma^w =< \$1, 1 >)\|$
$(\sigma^A =< 1, 4 > \ \&\ \sigma^w =< 1, 1 >)$

Figure 4.5 : Handling Constants

4.6.4 Array Accesses

When only part of an array is accessed in a statement, there will not be constraints on that array for that dimension. The sizes of the other variables will, however, be constrained by the size of the part of the array accessed. This includes subscripted arrays on the left-hand side.

If the size of the array access is defined in terms of the value of another variable the compiler needs to account for the fact that the value could make the access scalar. Figure 4.6 illustrates how the constraints are written to handle this situation. The first fields of σ^v do not appear in the actual constraint, but are left in to show the relationship of the other variables to the piece of v accessed.

$w = A * v(1 : j, :)$

would now show constraints:

$$
\begin{array}{l}
(\sigma^A =< 1,1 > (\& \ \sigma^{v(1:j,:)} =< 1,1 >) \ \& \ \sigma^w =< 1,1 >)\| \\
(\sigma^A =< 1,1 > (\& \ \sigma^{v(1:j,:)} =< j,\$1 >) \& \ \sigma^w =< j,\$1 >)\| \\
(\sigma^A =< \$1,\$2 > \ (\& \ \sigma^{v(1:j,:)} =< 1,1 >) \& \ \sigma^w =< \$1,\$2 >)\| \\
(\sigma^A =< \$1,j > (\& \ \sigma^{v(1:j,:)} =< j,\$2 >) \ \& \ \sigma^w =< \$1,\$2 >)\| \\
(\sigma^A =< 1,j > \ (\& \ \sigma^{v(1:j,:)} =< j,1 >) \& \ \sigma^w =< 1,1 >)
\end{array}
$$

Figure 4.6 : Dealing with Subscripted Array Access

Array accesses do give us some information about the size of the actual variable, however. The variable must be at least the size of the portion accessed in that dimension.[7] If the compiler can determine that the subscript size is greater than one, then it can add a constraint that forces the variable to be non-scalar, reducing the number of cliques.

4.6.5 Recombining Arrays

The reason redefining parts of an array is treated as defining an entirely new array is that different operations are performed based on whether the variable is a scalar or an array. Since arrays can grow from scalars in Matlab when redefined, this could be a

[7]This is only true because it is assumed that redefining parts of an array create an entirely new array in SSA.

problem in Fortran, as the operation would be invalid for an array. More importantly, if the compiler did not treat these as separate variables, the algorithm acts as though scalars and arrays conflict, which means a correct possibility could be overlooked by the algorithm. In this form of SSA arrays cannot grow. The SSA form is only required during analysis. After the analysis, the compiler can recombine two variables that were made separate in SSA if it does not change from a scalar to an array. However, the array may still have grown to a bigger array. The compiler must therefore make the array the maximum size of all the subscripted arrays. It must also keep track of which parts of the big array are used or defined in which operations.[8]

4.7 The Result

The compiler ends up with a set of possible variable type configurations over the analyzed procedure. For each possible configuration of sizes for the input parameters, the compiler creates an individual specialized and optimized variant. The appropriate variant will be linked directly with the user script at runtime.

Since procedure calls are treated as primitive operations, similar annotations to the hand-written annotations need to be generated and added to the database. These annotations are formed from the final information given by the algorithm. Each possible combination of sizes for input and output parameters allowed by at least one clique will be represented by the annotation for that subroutine. In essence, the compiler is computing type jump functions for interprocedural purposes.

4.8 Code Generation

To generate the optimized version, I use Sparse BLAS[26], or SpBlas, and the ATLAS-tuned BLAS[28] for now. There is enough functionality in these to handle the op-

[8] Arrays cannot shrink unless the entire array is redefined, in which case SSA treats the redefinition as a different variable

erations called by ArnoldiC, and the BLAS are specialized for the important cases for Arnoldi, except that the input `A` to Arnoldi will generally be sparse.[9] Therefore, operations on `A` are usually handled using SpBLAS. One of the advantages for using ATLAS is that the generated code will also automatically be optimized for the specific machine on which it is running, which is also a goal of telescoping languages. Other applications of telescoping languages may require more optimization than BLAS or SpBLAS provides. I leave this for future research.

[9]Only known after talking with the library writer. This is a case when the compiler would benefit from annotations.

Chapter 5

Inferring Intrinsic Types and Shapes

5.1 Intrinsic Types

The algorithm for intrinsic types is almost identical to the algorithm for size except for a few key issues. First, the constraints for intrinsic types are different since they operate on the intrinsic-type lattice. Also, instead of solving the equations for an exact type, the compiler only needs to find a range of possible intrinsic types. Slicing no longer helps, since intrinsic types need to be declared at the beginning of each control-flow block, so handling control flow will be a little different. Finally, subscripted accesses are no longer special cases.

5.1.1 Forming Constraints

The constraints on intrinsic types are similar to the constraints on size except that instead of working with infinite numbers (i.e. size could be any non-negative integer) the compiler operates on the intrinsic-type lattice. However, for size we only had two possibilities (scalar or array), and the equation solver handled finding actual values. For intrinsic types the lattice is bigger than the scalar versus array choice. Therefore, k (the number of constraints per level) is 6^p, where p is the number of parameters. However, in practical cases the number of possible intrinsic types that might work for a particular operation should be smaller than the entire lattice.

Also, the constraints will be tracking the range of possible intrinsic types for the variable on the lattice of possibilities. For example, an input argument that is defined as type real could actually be of type int when called. For the statement, $A = B + C$, Some of the constraints would be:

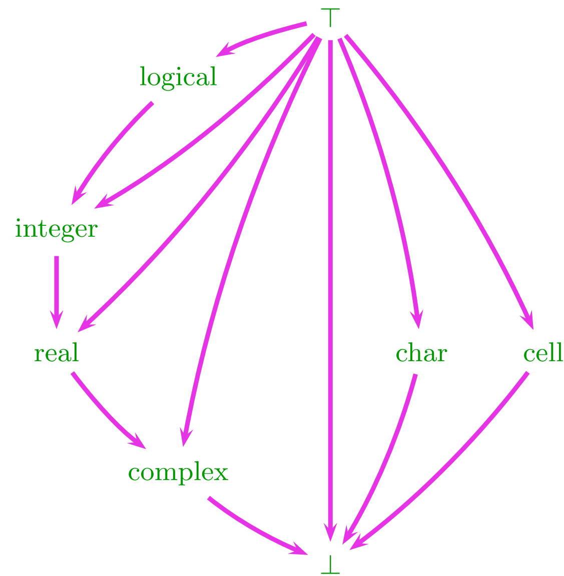

Figure 5.1 : Intrinsic-Type Lattice

...

$(int <= \tau^A <= int \ \& \perp <= \tau^B <= int \ \& \perp <= \tau^C <= int) \|$

$(real <= \tau^A <= real \ \& \perp <= \tau^B <= real \ \& \perp <= \tau^C <= real \|$

...

Two constraint clauses will conflict if a variable is in both clauses and the ranges of possibilities for the variable's intrinsic type do not intersect. The compiler still needs mutual exclusivity for the algorithm to run properly, so their ranges have to be defined finely enough for this property to hold.

Subscripted Arrays

The complications that arose from subscripted array accesses do not occur for intrinsic types since the compiler only wants to find the type over the array as a whole.

However, the compiler still treats defining a subscripted part of the array to be a redefinition of the whole array for purposes of SSA. Again, after the algorithm has been performed, the arrays can be merged back together with the bottom-most type. However, if the array is real in some parts of the code and complex in others it may be better to keep the arrays separate because of the overhead of dealing with complex values.

5.1.2 Solving the Equations

Once the compiler has found the cliques, solving the equations within the cliques corresponds to taking the intersection of all ranges for each variable in the clique.

5.1.3 Handling Control Flow

An intrinsic type cannot be defined by a value of a variable as it is determined only by the intrinsic type of the right-hand side. Therefore, the compiler only has to handle control flow. Slicing does not help here because declaration of variables occurs in the beginning before any operations can be performed. Therefore, at control-flow points the compiler must generate paths based on the outcome of the condition and, depending on the path, declare the ϕ variable to be whatever it would be on executing the path. This is especially useful if the ϕ variable could be either real or complex depending on the control flow. However, it is a cumbersome enough solution that in most cases it would be best to declare the ϕ node to be the meet of the possibilities (i.e. the bottom-most possible intrinsic type for definitions and the top-most for uses.)

5.2 Shape

Inferring shapes will be similar to inferring intrinsic types. For shapes, however, the compiler will need to rely much more heavily on the annotations. Because the compiler cannot generate a specialized variant for every possible shape, the compiler

will simply generate the most general cases unless it can infer that more specific cases are important. The compiler can infer this either from constraints on the procedure calls or from the user-defined annotations on the subroutine being analyzed. For this reason, if specific cases are possible and the library writer thinks that the cases are important enough to have an optimized variant based on the shape, the library writer must annotate the procedure with that possibility in order for the compiler to generate a specialized variant unless it can infer the same information from program operations.

The shape lattice includes all possible combinations of shape with top being the most general (dense) and bottom meaning that the shapes inferred could not be possible. Some examples of entries in the lattice include sparse, sparse banded, sparse symmetric, sparse symmetric banded, etc.

There are some BLAS routines specialized for different shapes (i.e. banded matrices), and SpBLAS handle sparse matrices. Beyond using these, generating optimized code for the different shapes is beyond the scope of this thesis.

Chapter 6

Experimental Evaluation

In order to evaluate ARGen, I performed the transformations and analysis on ArnoldiC by hand, as the ARGen system is not yet in working state.

The first transformation necessary was pruned SSA. As I transformed the code I also expanded it to have a single operation per line. While the compiler would not have to do this explicitly, it makes the analysis easier to follow when doing by hand. Figure 6.1 shows the transformed code. As it turned out, for AnoldiC, only one ϕ node is necessary. Therefore, ARGen is very efficient on ArnoldiC.

6.1 Analysis

The next step was to find constraints on the numbered statements from Figure 6.1. Figure 6.2 shows the hand-evaluated constraints on some of the statements within the loop. The fact that j can never be one is taken into account when forming the constraints.

The next steps were building the graph from the constraints by hand, finding the cliques, and solving for each of the cliques. The resulting possible configurations are shown as the columns in figure 6.3.

The configuration chosen depends entirely on σ^{A_1} and σ^{v_1}, which are both inputs.

Actually, the only possibilities on real applications in terms of size for ArnoldiC is the last column in figure 6.3. Annotations provided by the library writers could have narrowed down the possibilities to this single case by telling the compiler that the input A_1 is never scalar.

6.2 Experiments

I hand-translated ArnoldiC into Fortran based on the results of the last case to compare the runtimes between the Matlab code and the code ARGen will produce. The comparisons are shown in Figure 6.4. I included the runtime of ARPACK's dnaitr/dsaitr subroutine on the same matrix to show that the ARGen version has comparable runtime, although the ARPACK version behaves slightly differently than the Matlab code, since the ARPACK developers added functionality to the code when they hand-translated from Matlab to Fortran.[1] The outputs from the Matlab code and the ARGen code agreed.

I ran the experiments using an array from Matrix Market[19] for the input parameter A_1. The array was a 362×362 real, sparse symmetric matrix with 3074 entries. I set k_1 to 30 and v_1 to be a 362-length vector. I used version 6.1 for Matlab, which also uses ATLAS-tuned BLAS at the bottem level. The runtimes were measured on a 143 MHz SPARC processor.

I represented this array in both sparse-coordinate and dense formats to compare the different codes on both sparse and dense matrices. The Matlab * operation is overloaded for both sparse and dense matrices, so I did not need to change the Matlab ArnoldiC to compare the different forms. For the ARGen code, I used SpBLAS to compute both of the `w=A*v` operations for the sparse format, and dgemv/dsymv from the ATLAS tuned BLAS for the dense format. I used the same routines to provide `w=A*v` to the ARPACK code through the reverse communication interface. By comparing the different representations of the same Matrix, I am able to show how user annotations can play an important role in generating efficient code, since the fact that A_1 should almost always be sparse could only be inferred from the library writers. Representing the matrix in sparse coordinate form greatly improved

[1]Because of the descrepency between the Matlab and Fortran ARPACK code, I was not able to show ARGen producing the Fortran ARPACK in it's entirety. However, ARGen would be able to generate a Fortran ARPACK equivalent given equivalent Matlab code.

the runtime, especially for Matlab.

Since the matrix I used is also symmetric, I represented the matrix both as a symmetric matrix and a nonsymmetric matrix (having all entries) to determine the benefits of optimizing for shape. Matlab does not make this distinction at the top level since it expects all matrices to have all elements represented explicitly. However, at the function level, it does distinguish between symmetric and nonsymmetric matrices. In the ARGen code, I changed `w=A*v` to be dsymv for symmetric matrices and to dgemv for nonsymmetric matrices in the dense code. For the sparse code, I changed the description of the matrix passed to the SpBLAS. I used the same routines for the same cases in ARPACK, but called either dnaitr or dsaitr depending on whether I used the nonsymmetric form or the symmtetric form. Using the symmetric routines improved the performance slightly on the dense representations, but less on the sparse. Overall, there is a 50% improvement in speed from representing the matrix as sparse and symmetric as opposed to representing it as dense and non-symmetric.

I also represented the array as complex to show the benefit of accurately inferring intrinsic types. There is a 54% speed improvement from representing the array as real versus representing it as complex.

Finally, the results show that there is benefit to be gained from moving the Matlab code to Fortran even though Matlab 6.1 uses ATLAS tuned BLAS as well. Also, ARGen was able to achieve better performance than the corresponding ARPACK code. This could be due, in part, to the added functionality in the ARPACK code. However, the primary reason ARPACK ran slower is due to the fact that the XYaitr subroutines handle two problems. An inputed flag determined which problem was solved. This meant that there was a lot of extra control flow in the subroutine. Since the subroutine was called inside a loop, this affected performance. Also, the reverse communication added some overhead, which would probably not be apparent on a larger matrix. Since ARGen is automatically generating specialized variants, it would generate different subroutines for the different problems avoiding this overhead.

```
function[V, H, f] = ArnoldiC(A_1, k_1, v_1);
1       v_2 = v_1/norm(v_1);
2       w_1 = A_1 * v_2;
3       α_1 = v_2' * w_1;
4       temp_1 = v_2 * α_1;
5       f_1 = w_1 − temp_1;
6       c_1 = v_2' * f_1;
7       temp_2 = v_2 * c_1;
8       f_2 = f_1 − temp_2;
9       α_2 = α_1 + c_1;
10      V_1(:, 1) = v_2;
11      H_1(1, 1) = α_2;
12      for j = 2 : k_1,
            f_3 = φ(f_2, f_5);
13          β_1 = norm(f_3);
14          v_3 = f_3/β_1;
15          H_2(j, j − 1) = β_1;
16          V_2(:, j) = v_3;
17          w_2 = A_1 * v_3;
18          h_1 = V_2(:, 1 : j)' * w_2;
19          temp_3 = V_2(:, 1 : j) * h_1;
20          f_4 = w_2 − temp_3;
21          c_2 = V_2(:, 1 : j)' * f_4;
22          temp_4 = V_2(:, 1 : j) * c_2;
23          f_5 = f_4 − temp_4;
24          h_2 = h_1 + c_2;
25          H_3(1 : j, j) = h_2;
        end
```

Figure 6.1 : Pruned SSA Form of ArnoldiC

13 $\sigma^{\beta_1} =<1,1> \; \& \; \sigma^{f_3} =<1,1> \mid$

$\sigma^{\beta_1} =<1,1> \; \& \; \sigma^{f_3} =<\$21,\$22>$

14 $\sigma^{v_3} =<1,1> \; \& \; \sigma^{f_3} =<1,1> \; \sigma^{beta_1} =<1,1> \mid$

$\sigma^{v_3} =<1,1> \; \& \; \sigma^{f_3} =<1,\$23> \; \& \; \sigma^{\beta_1} =<1,\$23> \mid$

$\sigma^{v_3} =<1,\$23> \; \& \; \sigma^{f_3} =<1,1> \; \& \; \sigma^{\beta_1} =<\$23,1> \mid$

$\sigma^{v_3} =<\$23,\$24> \; \& \; \sigma^{f_3} =<\$23,\$24> \; \& \; \sigma^{\beta_1} =<1,1> \mid$

$\sigma^{v_3} =<\$23,\$24> \; \& \; \sigma^{f_3} =<\$23,\$25> \; \& \; \sigma^{\beta_1} =<\$24,\$25>$

15 $\sigma^{beta_1} =<1,1>$

16 $\sigma^{V_2} =<1,> \; \& \; \sigma^{v_3} =<1,1> \mid$

$\sigma^{V_2} =<\$26,> \; \& \; \sigma^{v_3} =<\$26,1>$

17 $\sigma^{w_2} =<1,1> \; \& \; \sigma^{A_1} =<1,1> \; \& \; \sigma^{v_3} =<1,1> \mid$

$\sigma^{w_2} =<1,1> \; \& \; \sigma^{A_1} =<1,\$27> \; \& \; \sigma^{v_3} =<\$27,1> \mid$

$\sigma^{w_2} =<\$27,\$28> \; \& \; \sigma^{A_1} =<1,1> \; \& \; \sigma^{v_3} =<\$27,\$28> \mid$

$\sigma^{w_2} =<\$27,\$28> \; \& \; \sigma^{A_1} =<\$27,\$28> \; \& \; \sigma^{v_3} =<1,1> \mid$

$\sigma^{w_2} =<\$27,\$28> \; \& \; \sigma^{A_1} =<\$27,\$29> \; \& \; \sigma^{v_3} =<\$29,\$28>$

18 $\sigma^{h_1} =<j,\$30> \; \& \; \sigma^{V_2} =<\$30,> \; \& \; \sigma^{w_3} =<1,1> \mid$

$\sigma^{h_1} =<j,\$30> \; \& \; \sigma^{V_2} =<\$31,> \; \& \; \sigma^{w_3} =<\$31,\$30>$

19 $\sigma^{temp_3} =<1,1> \; \& \; \sigma^{V_2} =<1,> \; \& \; \sigma^{h_1} =<j,1> \mid$

$\sigma^{temp_3} =<\$32,j> \; \& \; \sigma^{V_2} =<\$32,> \; \& \; \sigma^{h_1} =<1,1> \mid$

$\sigma^{temp_3} =<\$32,\$33> \; \& \; \sigma^{V_2} =<\$32,> \; \& \; \sigma^{h_1} =<j,\$33> \mid$

20 $\sigma^{f_4} =<1,1> \; \& \; \sigma^{w_2} =<1,1> \; \& \; \sigma^{temp_3} =<1,1> \mid$

$\sigma^{f_4} =<\$34,\$35> \; \& \; \sigma^{w_2} =<1,1> \; \& \; \sigma^{temp_3} =<\$34,\$35> \mid$

$\sigma^{f_4} =<\$34,\$35> \; \& \; \sigma^{w_2} =<\$34,\$35> \; \& \; \sigma^{temp_3} =<1,1> \mid$

$\sigma^{f_4} =<\$34,\$35> \; \& \; \sigma^{w_2} =<\$34,\$35> \; \& \; \sigma^{temp_3} =<\$34,\$35> \mid$

21 $\sigma^{c_2} =<j,\$36> \; \& \; \sigma^{V_2} =<\$36,> \; \& \; \sigma^{f_4} =<1,1> \mid$

$\sigma^{c_2} =<j,\$36> \; \& \; \sigma^{V_2} =<\$37,> \; \& \; \sigma^{f_4} =<\$37,\$36>$

22 $\sigma^{temp_4} =<1,1> \; \& \; \sigma^{V_2} =<1,> \; \& \; \sigma^{c_2} =<j,1> \mid$

$\sigma^{temp_4} =<\$38,j> \; \& \; \sigma^{V_2} =<\$38,> \; \& \; \sigma^{c_2} =<1,1> \mid$

$\sigma^{temp_4} =<\$38,\$39> \; \& \; \sigma^{V_2} =<\$38,> \; \& \; \sigma^{c_2} =<j,\$39> \mid$

23 $\sigma^{f_5} =<1,1> \; \& \; \sigma^{f_4} =<1,1> \; \& \; \sigma^{temp_4} =<1,1> \mid$

$\sigma^{f_5} =<\$40,\$41> \; \& \; \sigma^{f_4} =<1,1> \; \& \; \sigma^{temp_4} =<\$40,\$41> \mid$

$\sigma^{f_5} =<\$40,\$41> \; \& \; \sigma^{f_4} =<\$40,\$41> \; \& \; \sigma^{temp_4} =<1,1> \mid$

$\sigma^{f_5} =<\$40,\$41> \; \& \; \sigma^{f_4} =<\$40,\$41> \; \& \; \sigma^{temp_4} =<\$40,\$41> \mid$

24 $\sigma^{h_2} =<1,1> \; \& \; \sigma^{h_1} =<1,1> \; \& \; \sigma^{c_2} =<1,1> \mid$

$\sigma^{h_2} =<\$42,\$43> \; \& \; \sigma^{h_1} =<1,1> \; \& \; \sigma^{c_2} =<\$42,\$43> \mid$

$\sigma^{h_2} =<\$42,\$43> \; \& \; \sigma^{h_1} =<\$42,\$43> \; \& \; \sigma^{c_2} =<1,1> \mid$

$\sigma^{h_2} =<\$42,\$43> \; \& \; \sigma^{h_1} =<\$42,\$43> \; \& \; \sigma^{c_2} =<\$42,\$43> \mid$

25 $\sigma^{h_2} =<j,1>$

Figure 6.2 : Constraints on Statements within the Loop

σ^{A_1}	$<1, 1>$	$<1, 1>$	$<\$1, \$1>$
σ^{v_1}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{k_1}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{v_2}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{w_1}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{α_1}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{f_1}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{c_1}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{f_2}	$<1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{α_2}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{V_1}	$<1,>$	$<\$1,>$	$<\$1,>$
σ^{f_3}	$<\$1, \$1>$	$<\$1, \$1>$	$<\$1, \$1>$
σ^{β_1}	$<1, 1>$	$<1, 1>$	$<1, 1>$
σ^{v_3}	$<\$1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{V_2}	$<\$1,>$	$<\$1,>$	$<\$1,>$
σ^{w_2}	$<\$1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{h_1}	$<j, 1>$	$<j, 1>$	$<j, 1>$
σ^{f_4}	$<\$1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{c_2}	$<j, 1>$	$<j, 1>$	$<j, 1>$
σ^{f_5}	$<\$1, 1>$	$<\$1, 1>$	$<\$1, 1>$
σ^{h_2}	$<j, 1>$	$<j, \$1>$	$<j, 1>$

Figure 6.3 : Configurations Satisfying Cliques

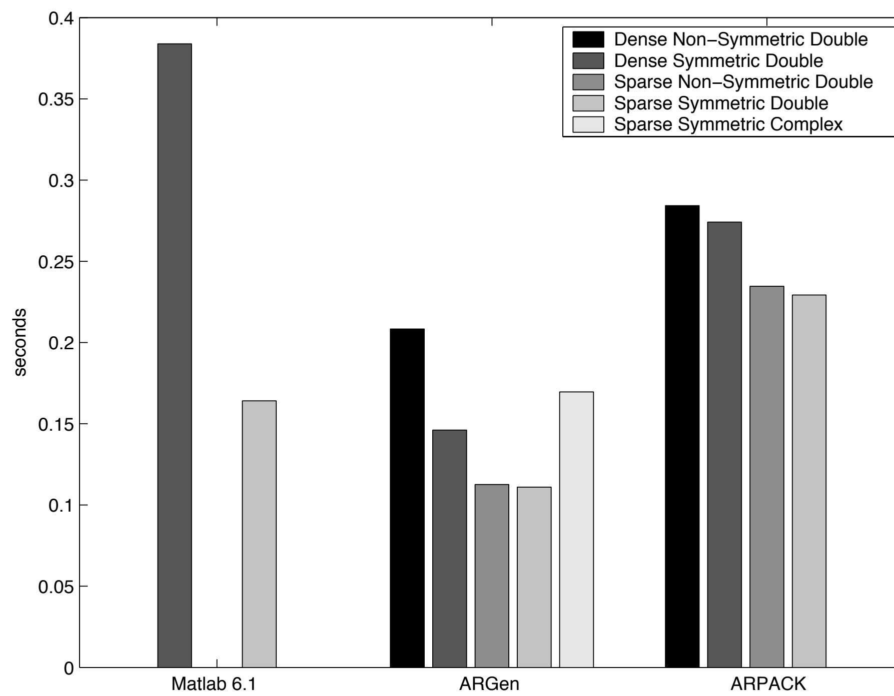

Figure 6.4 : Comparison of Arnoldi Subroutines

Chapter 7

Related Work

7.1 Work in Translating Matlab

MathWorks has a Matlab compiler called mcc, which translates the Matlab source code into C[18]. However, the C code just makes calls to library subroutines which handle the operations on the arrays. No type analysis or other interprocedural analysis is performed at this level.

The FALCON Project at the University of Illinois is also aimed at aiding in the development of high-performance applications by allowing users to code in Matlab.[9, 10] The FALCON project performs translation from Matlab into FORTRAN 90. However, the FALCON project does not follow the telescoping languages philosophy, in that it handles all procedure calls by inlining. This leads to long script-compilation times for optimization. FALCON does perform some limited type inferencing to achieve its goals. However, it only performs forward propagation with one backward propagation step. The definition of type used in this thesis was motivated by the FALCON project.

MaJIC, also at the University of Illinois, builds on the FALCON project.[1, 17] MaJIC performs ahead-of-time speculative analysis in which it performs type inferencing with limited backward propagation. This speculative analysis is meant to aid the just-in-time compiler. Therefore, the type inferencing required does not need to be as complete as what is required to precompile for all possible calling contexts. Menon and Pingali explore source-level optimization on Matlab code in the context of MaJIC[20, 21].

There are several projects underway for translating Matlab to lower-level lan-

guages. These include the Otter system at Oregon State University, the CONLAB compiler from University of Umea in Sweden, and Menhir from Irisa in France [24, 12, 5]. The MATCH project at Northwestern University attempts to compile Matlab directly to special purpose hardware [23].

7.2 Type Inferencing in the Programming Languages Community

Type Inferencing for functional languages has been studied extensively by the programming languages community. However, its focus is different from the telescoping languages strategy. Programming-languages type-inferencing methods concentrate on proving programs correct and on providing feedback to the programmers to aid them in debugging and understanding the behavior of the code. Telescoping languages, on the other hand, takes an almost opposite point of view. Telescoping languages assumes the code is correct, and relies on the programmer to provide annotations, which will aid the compiler in understanding the possible calling contexts for the code.

Matlab is an imperative language geared towards working with arrays. Most of the research in type inferencing from the programming languages community deals only with functional languages and is not concerned with handling arrays.

Hindley-Milner type systems, well-known examples of type inferencing systems from the programming languages community, use a constraint-based representation to aid in inferring types[22], although the constraints are formulated differently. These systems can be solved using unification-based algorithms. One advantage to the Hindley-Milner systems is that they can describe polymorphism in languages. Unfortunately, pure polymorphism is not sufficient to express Matlab's type system[3]. Also, unification produces the most general possible types for the polymorphic values. This does not give enough information to generate all possible variants that might occur in specific calling contexts. Moreover, since telescoping languages gener-

ates optimized variants of each possible calling context, the tightest information on combinations of possible types is required.

Cormac Flanagan used componential set-based analysis to infer types for the purposes of debugging[13]. However, his system will not handle the heavily overloaded operators found in Matlab. Also, like the Hindley-Milner systems, the outcome of his analysis would be too general for the purposes of telescoping languages.

7.3 Constraints Logic Programming

Constraint Logic Programming extends syntactic logic programming (i.e., unification) with semantic constraints over specific domains. Inferring types using constraints over the type domain can be considered as an example of constraint logic programming. There are several general systems designed to solve constraint logic programming problems quickly, including CHIP [11], CLP($\mathcal{R}$) [14], Prolog-III [6], and ECL^{i}PS^{e} [27]. However, the algorithm described in this thesis is designed to take full advantage of the specific aspects of this problem to reduce runtime. In the general system, the bounds on the runtime would not be guaranteed.

Chapter 8

Conclusion

Telescoping languages seeks to aid scientists in producing high-performance code by allowing them to use a high-level scripting language, such as Matlab. It calls for an extensive library compilation phase in which the code is translated to lower level languages for which highly tuned vendor compilers already exist. Specialized variants of the code are created based on possible calling contexts inferred from the library itself or hinted at by the library writers.

ARPACK is a useful and relatively simple example of where telescoping-language technology can be applied. It was written in Matlab and then hand-translated into Fortran 77 for performance. The Fortran library has specialized variants of each subroutine based on type, showing that optimizing for specific types is useful and necessary. However, type specialization is limited by the energy of the creators. By automating this process, even more specialized variants could be created for more refined types.

Because Matlab is weakly typed, type inferencing is necessary for code generation and specialization. Type information flows both forward and backward, but the same information can be found by analyzing the whole procedure simultaneously. This thesis provided an efficient algorithm for doing this.

This thesis demonstrates an efficient, general algorithm for inferring types in high-level code and, using the information, develops a system (still in the implementation phase), ARGen, which is powerful enough to generate an equivalent of the Fortran ARPACK from the Matlab code.

Bibliography

[1] Goerge Almási and David Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.

[2] http://www.caam.rice.edu/software/ARPACK/. ARPACK.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[4] Arun Chauhan. Telescoping matlab for dsp applications. Thesis Proposal.

[5] Stéphane Chauveau and François Bodin. MENHIR: An environment for high performance MATLAB. In *Proceedings of the Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 1998.

[6] Alain Colmerauer. An introduction to Prolog-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[7] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, Atlanta, GA, June 1988.

[8] Thomas Corman, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. The MIT Press/McGraw Hill, 1990.

[9] Luiz DeRose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, March 1999.

[10] Luiz Antônio DeRose. *Compiler Techniques for Matlab Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[11] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and François Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, December 1988.

[12] Peter Drakenberg, Peter Jacobson, and Bo Kågström. A CONLAB compiler for a distributed memory multicomputer. In *SIAM Conference on Parallel Processing for Scientific Computing*, volume 2, pages 814–821, 1993.

[13] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

[14] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992. Also available as Technical Report, IBM Research Division, RC 16292 (#72336), 1990.

[15] Ken Kennedy. Telescoping Languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proceedings of International Parallel and Distributed Processing Symposium*, May 2000.

[16] Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnson, John Mellor-Crummey, and Linda Torczon. Telescoping Languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.

[17] http://polaris.cs.uiuc.edu/majic/. MAJIC Project.

[18] http://www.mathworks.com/. Mathworks, Inc.

[19] http://math.nist.gov/MatrixMarket/. Matrix Market.

[20] Vijay Menon and Keshav Pingali. A case for source level transformations in Matlab. In *Proceedings of the ACM SIGPLAN / USENIX Conference on Domain Specific Languages*, 1999.

[21] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical code. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, 1999.

[22] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17(2):348–375, 1978.

[23] Anshuman Nayak, Malay Haldar, Abhay Kanhere, Pramod Joisha, Nagraj Shenoy, Alok Choudhary, and Prith Banerjee. A library-based compiler to execute MATLAB programs on a heterogeneous platform. In *Proceedings of the Conference on Parallel and Distributed Computing Systems*, August 2000.

[24] Michael J. Quinn, Alexey Malishevsky, and Nagajagadeswar Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, August 1998.

[25] Dan C. Sorensen, Richard B. Lehoucq, and Chao Yang. *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, 1997.

[26] http://math.nist.gov/spblas/. SpBLAS.

[27] Mark Wallace, Stefano Novello, and Joachim Schimpf. *ECL^iPS^e: A Platform for Constraint Logic Programming*. William Penney Laboratory, Imperial College, London, 1997.

[28] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of SC: High Performance Networking and Computing Conference*, November 1998.